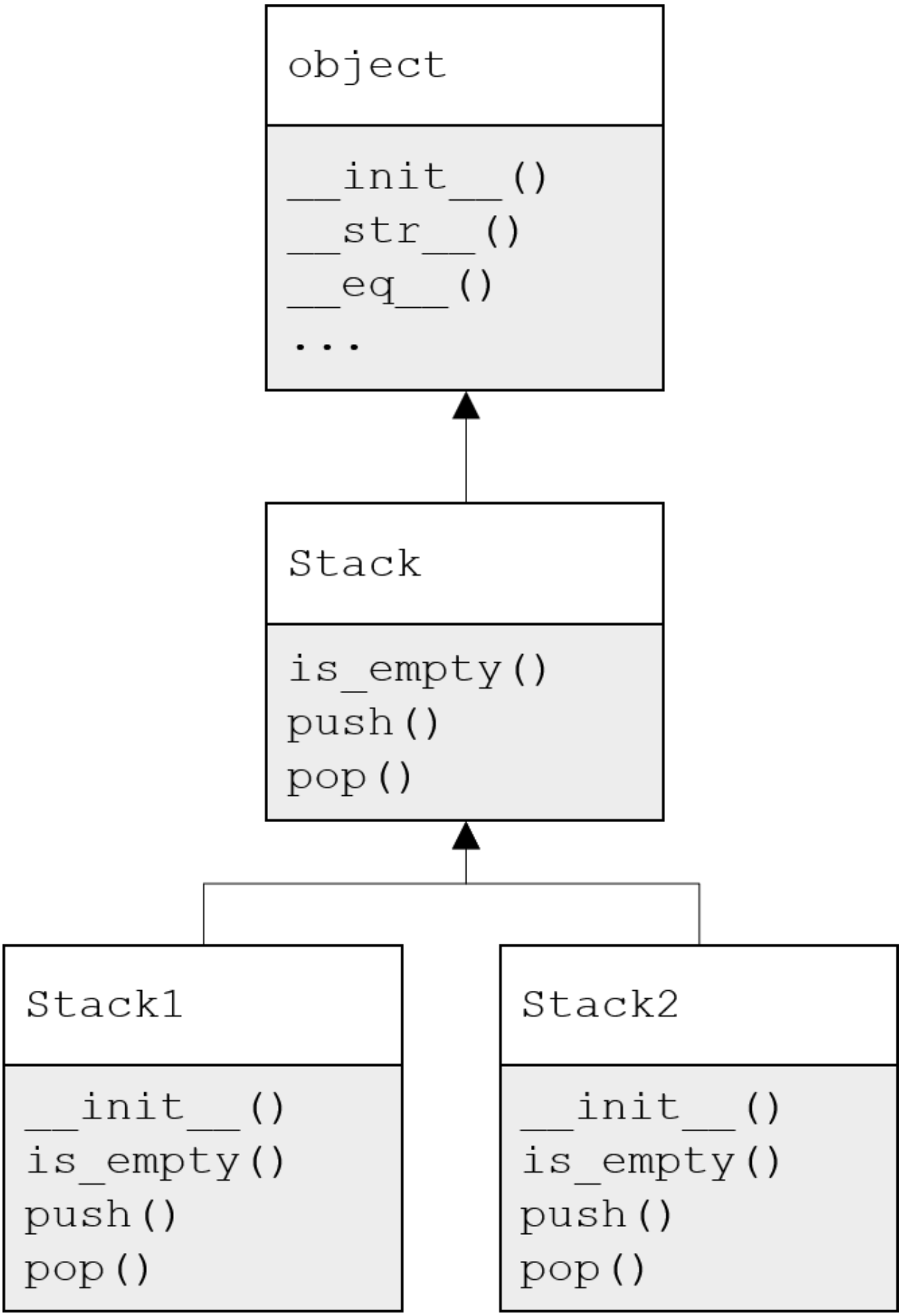
# 9.9 The `object` Superclass

In our very first chapter, we described every piece of data as an *object*, and have continued to use this term throughout this course. It turns out that "object" is not merely a theoretical concept, but made explicit in the Python language. Python has a special class called `object`, which is an *ancestor class*[1] of every other class, both built-in classes like `int` or our custom

[1] By "ancestor" we mean either a parent class, or a parent of a parent class, etc.

data classes and the classes we've defined in this chapter. And this includes abstract classes like `Stack`!

By default, whenever we define a new class (including data classes), if we do not specify a superclass in parentheses, `object` is the *implicit* superclass, which is why we can write `class Stack:` instead of `class Stack(object):`.




*The `object` special methods*

This `object` class defines several special methods as part of its shared public interface, including:[2]

[2] The Python convention is to name methods that have a special purpose with double underscores. These are sometimes called "dunder" methods (**d**ouble **under**score).

- `__init__(self, ...)`, the initializer
- `__str__(self)`, which returns a `str` representation of the object.

### *Method inheritance*

Unlike our `Stack` abstract class earlier this chapter, the `object` class is actually *not abstract*, and implements each of these methods. We can use this to illustrate a different form of inheritance, where the superclass is a concrete class. In this case, inheritance is used not just to define a shared public interface, but also to provide *default implementations* for each method in the interface.

For example, suppose we create a dummy class with a completely empty body:

```
class Donut:
    """A donut, because why not?"
```

This class inherits the `object.__init__` method, which allows us to create new `Donut` instances.

```
>>> donut = Donut()
>>> type(donut)
<class '__main__.Donut'>
```

Similarly, this class inherits the `object.__str__` method, which returns a string that states the class name and memory location of the object:

```
>>> d = Donut()
>>> d.__str__()
'<__main__.Donut object at 0x7fc299d7b588>'
```

We can use the built-in `dir` function to see all of the special methods that `Donut` has inherited from `object`:[3]

[3] Though this list includes few special attributes set directly by the Python interpreter, which are beyond the scope of this course.

```
>>> dir(Donut)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
         '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
         '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
         '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
         '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
         '__weakref__']
```

There is another reason these methods are special beyond simply being inherited from the `object` superclass: they are often called by other functions or parts of Python syntax. For example, we have already seen how the `__init__` method is called when a new object is initialized.

The `__str__` method is called when we attempt to convert an object to a string by calling `str` on it:

```
>>> d = Donut()
>>> d.__str__()
'<__main__.Donut object at 0x7fc299d7b588>'
>>> str(d)
'<__main__.Donut object at 0x7fc299d7b588>'
```

Similarly, the built-in `print` function actually first converts its arguments into strings using their `__str__` methods, and then prints out the resulting text.

### *Method overriding*

Now, even though the `object` superclass contains default implementations of `__init__` and `__str__`, in practice we often want to define our own custom implementations of these methods.

Every time we've defined our own `__init__` in a class, we have **overridden** the `object.__init__` method. Formally, we say that a class `C` **overrides** a method `m` when the method `m` is defined in the superclass of `C`, and is also given a concrete implementation in the body of `C`.[4]

[4] This definition applies whether the superclass of `C` has `m` as an abstract or concrete method. For example, we could say that `Stack1` overrides the `push` and `pop` method from its abstract superclass `Stack`.

Similarly, when we defined a custom exception class in Section 9.5,

```
class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'pop may not be called on an empty stack'
```

this class *overrode* the `__str__` method to use its own string representation, which is displayed when this exception is raised.